Inheritance & Polymorphism

Week 7
Gaddis: Chapter 15

CS 5301
Fall 2018
Jill Seaman

Inheritance

• A way to create a new class from an existing class
• The new class is a specialized version of the existing class
• Base class (or parent) – the existing class
• Derived class (or child) – inherits from the base class
• The derived class contains all the members from the base class (in addition to the ones in the derived class).

```
class Student {
    . . .
}

class UnderGrad : public Student {
    . . .
}  // Base class

}  // Derived class
```

Access to private members

- **private members:**
  - char letter;
  - float score;
  - void calcGrade();
- **public members:**
  - void setScore(float);
  - float getScore();
  - char getLetter();

When Test class inherits from Grade class using public class access, it looks like this:

```
class Grade
{
private members:
    int numQuestions;
    float pointsEach;
    int numMissed;
public members:
    Test(int, int);
}

class Test : public Grade
{
private members:
    int numQuestions;
    float pointsEach;
    int numMissed;
public members:
    Test(int, int);
    void setScore(float);
    float getScore();
    char getLetter();
}
```

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created,
  1. the base class’s (default) constructor is executed first,
  2. followed by the derived class’s constructor
- When an object of a derived class is destroyed,
  1. the derived class destructor is called first,
  2. then the base class destructor

An instance of Test contains letter and score, but they are not directly accessible from inside (or outside) the Test member functions.
Constructors and Destructors:

```cpp
class BaseClass {
public:
    BaseClass() {
        cout << "This is the BaseClass constructor.\n"; }
    ~BaseClass() {
        cout << "This is the BaseClass destructor.\n"; }
};
class DerivedClass : public BaseClass {
public:
    DerivedClass() {
        cout << "This is the DerivedClass constructor.\n"; }
    ~DerivedClass() {
        cout << "This is the DerivedClass destructor.\n"; }
};
int main() {
    cout << "We will now define a DerivedClass object.\n";
    DerivedClass object;
    cout << "The program is now going to end.\n";
}
```

Output:
```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
```

Redefining Base Class Functions

- **Redefining function**: a function in a derived class that has the same name and parameter list as a function in the base class
- Not the same as overloading – with overloading, parameter lists must be different
- Objects of base class use base class version of function; objects of derived class use derived class version of function.
- To call the base class version from the derived class version, you must prefix the name of the function with the base class name and the scope resolution operator:

```cpp
 Rectangle::display()
```

Passing Arguments to a non-default
Base Class Constructor

- Allows programmer to choose which base class constructor is called from the derived class constructor
- Specify arguments to base constructor in the derived constructor function header:

```cpp
//assuming Square is derived from Rectangle:
Rectangle::Rectangle(double w, double len) {
    width = w; length = len; }
Square::Square(int side) : Rectangle(side, side) {
    // code for Square constr goes here, if any }
```

- **You must** specify a call to a base class constructor if base class has no default constructor

```cpp
int main() {
    DerivedClass object;
    cout << "This is the DerivedClass destructor.\n";
    cout << "This is the BaseClass destructor.\n";
}
```

Redefining Base Class Functions:

```cpp
class Animal {
private:
    string species;
public:
    Animal()  { species = "Animal"; }
    Animal(string spe) { species = spe; }
    void display() { cout << "species " << species; }
};
class Primate: public Animal {
private:
    int heartCham;
public:
    Primate() : Animal("Primate") { }
    Primate(int in) : Animal("Primate")
            { heartCham = in; }
    void display() {
        Animal::display(); // calls base class display()
        cout << ", \# of heart chambers " << heartCham;
    }
};
```
Include Guards

```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
#endif
```

- These preprocessor directives prevent the header file from accidentally being included more than once.
- Required when a base class has 2 derived classes, and both derived classes are included in a driver.

Polymorphism

- The Greek word poly means many, and the Greek word morphism means form.
- So, polymorphism means 'many forms'.
- In object-oriented programming (OOP), polymorphism refers to:
  - identically named (and redefined) functions
  - that have different behavior depending on the (specific derived) type of object they are called on.

Example of polymorphism?

```cpp
class Animal {
    private:
        ...
    public:
        void speak() { cout << "none " ; }
};
class Cat : public Animal {
    private:
        ...
    public:
        void speak() { cout << "meow " ; }
};
class Dog : public Animal {
    private:
        ...
    public:
        void speak() { cout << "bark " ; }
};
```

- IF the output is "meow bark", yes, polymorphism.
  - The behavior of `a` in `f` would depend on its specific (derived) type.
- IF the output is "none none", no it's not.

Polymorphism in C++

- Polymorphism in C++ is supported through:
  - virtual functions AND
  - pointers to objects OR reference parameters.
- without these, C++ determines which function to invoke at compile time (using the variable type).
- when virtual functions and pointer/references are used together, C++ determines which function to invoke at run time (using the specific type of the instance currently referenced by the variable).
Virtual functions

- **Virtual member function**: function in a base class that expects to be redefined in derived class
- Function defined with key word *virtual*:
  ```cpp
  virtual void Y() {...}
  ```

- Supports **dynamic binding**: functions bound at run time to function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Example virtual functions

```cpp
class Animal {
public:
  virtual void speak();
  int age();
};
class Cat : public Animal {
public:
  virtual void speak(); // redefining a virtual
  int age();            // redefining a normal function
};
int main()
{
  Cat morris;
  Animal *pA = &morris;  // using a pointer to get dynamic binding
  pA -> age();   // Animal::age() is invoked (base) (not virtual)
  pA -> speak(); // Cat::speak() is invoked (derived)
...
}
```

Heterogeneous Array version 1:

```cpp
class COne {
public:
  void vWhoAmI() { cout << "I am One" << endl; }
};
class CTwo : public COne {
public:
  void vWhoAmI() { cout << "I am Two" << endl; }
};
class CThree : public CTwo {
public:
  void vWhoAmI() { cout << "I am Three" << endl; }
};
int main()
{
  (COne *)apCOne[3] = { new COne, new CTwo, new CThree };
  for (int i = 0; i < 3; i++)
    apCOne[i] -> vWhoAmI();
}
```

```cpp
void f (Animal &a) {
  a.speak();
}
int main() {
  Cat c;
  Dog d;
  f(c);
  f(d);
}
```
Heterogeneous Array version 2:

```cpp
class COne {
    public:
        virtual void vWhoAmI() { cout << "I am One" << endl; }
};

class CTwo : public COne {
    public:
        void vWhoAmI() { cout << "I am Two" << endl; }
};

class CThree : public CTwo {
    public:
        void vWhoAmI() { cout << "I am Three" << endl; }
};

int main() {
    COne *apCOne[3] = { new COne, new CTwo, new CThree };
    for (int i = 0; i < 3; i++)
        apCOne[i] -> vWhoAmI();
}
```

Output:
```
I am One
I am Two
I am Three
```

Abstract classes and Pure virtual functions

- **Pure virtual function**: a virtual member function that **must** be overridden in a derived class.
- The `= 0` indicates a pure virtual function
- Must have no function definition in the base class.

Abstract base class: a class that can have no objects (instances).
- Serves as a basis for derived classes that will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function.

Example: Abstract Class

```cpp
class CShape {
    public:
        CShape ( ) { }
        virtual void vDraw ( ) const = 0; // pure virtual function
};

CShape CShape1; // Error: object of abstract class
CShape* pCShape; // Ok
CShape CShapeFun(); // Error: return type
void vg(CShape); // Error: argument type
```
Example: Abstract Class

- Pure virtual functions are inherited as pure virtual functions.

```cpp
class CAbstractCircle : public CShape {
private:
   int m_iRadius;
public:
   void vRotate(int) {}
   // CAbstractCircle ::vDraw() is a pure virtual function
};
```

- Or else:

```cpp
class CCircle : public CShape {
private:
   int m_iRadius;
public:
   void vRotate(int) {}
   void vDraw();  // define here or in impl file
};
```

Heterogeneous collection: abstract base class

```cpp
class Animal {
private:
   string name;
public:
   Animal(string n) {name = n;}
   virtual void speak() = 0;
};
class Cat : public Animal {
public:
   Cat(string n) : Animal(n) {}
   void speak() {cout << "meow ";}
};
class Dog : public Animal {
public:
   Dog(string n) : Animal(n) {}
   void speak() {cout << "bark ";}
};
class Pig : public Animal {
public:
   Pig(string n) : Animal(n) {}
   void speak() {cout << "oink ";}
}
```

```cpp
int main()
{
   Animal* animals[7] = {
      new Cat("Charlie"),
      new Cat("Scamp"),
      new Dog("Penny"),
      new Cat("Libby"),
      new Cat("Patches"),
      new Dog("Milo"),
      new Pig("Wilbur")
   };
   for (int i=0; i<7; i++) {
      animals[i]->speak();
   }
}
```

Output:
```
meow meow bark meow meow bark oink
```